University of Colorado, Boulder
CU Scholar

Computer Science Technical Reports | Computer Science

Spring 5-1-1998

ML-DEWS: A Workflow Change Specification Model and Language ; CU-CS-870-98

Clarence A. Ellis
University of Colorado Boulder

Karim Keddara
University of Colorado Boulder

Follow this and additional works at: http://scholar.colorado.edu/csci_techreports

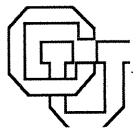
Recommended Citation
Ellis, Clarence A. and Keddara, Karim, "ML-DEWS: A Workflow Change Specification Model and Language ; CU-CS-870-98" (1998). *Computer Science Technical Reports*. 818.
http://scholar.colorado.edu/csci_techreports/818

This Technical Report is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.

ML-DEWS: A Workflow Change Specification Model and Language

Clarence Ellis
Karim Keddara

CU-CS-870-98



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

$\mathcal{ML}$-$\mathcal{DEWS}$: A Workflow Change Specification Model and Language

Clarence Ellis and Karim Keddara

University of Colorado,
CTRG Labs,
Dept of Computer Science,
Boulder CO 80309-0430, USA

Abstract. Organizations that are geared for success within today's business environments must be capable of rapid and continuous change. Dynamic change is a large and pervasive problem which surfaces within organizational workflows as well as within software engineering, manufacturing, and numerous other domains. Procedural changes, performed in an ad hoc manner, can cause inefficiencies, inconsistencies, and catastrophic breakdowns within organizations. This document is concerned with change, especially dynamic change, to organizational procedures. We explain a taxonomy of change modalities, and present a modeling language for the unambiguous specification of procedural change. This language, called $\mathcal{ML}$-$\mathcal{DEWS}$, complements the formal model of dynamic change previously presented by the authors, but is actually independent of the particular model used to describe organizational workflows. Issues of exception handling, temporal specification, and participatory change are conveniently handled within the framework presented in this document.

Keywords: dynamic procedural change, case migration, change modalities.

1 Introduction

A 1998 issue of Information Systems News featured an article headlined "Organizations Which Cannot Dynamically Change Cannot Survive." This is now a well-known theme in the business literature. The article notes that there are many kinds of change (e.g. organizational, social, ...) and that change is often done in an ad hoc manner. Thus, the ramifications of change, particularly complex changes within large organizations, are frequently not well understood, sometimes resulting in surprising negative side effects.

Within the Collaboration Technology Research Group (CTRG), we have been studying and analyzing issues of organizational change and dynamic change for a number of years. We firmly believe the thesis put forth in this document that change itself should be viewed as a process. To this end, we present a model and a language to describe the change process. The model and language are specifically concerned with procedural change within workflow systems. This model is, in fact, a meta-model because it facilitates the the examination and change of workflow schemas. It represents a structure within which we can analyze notions of temporal evolution, dynamic change, and exceptions. In previous publications we have presented formal definitions of dynamic change, and verified change correctness properties [9]. The work reported within this document also forms a portion of the PhD thesis of author Keddara.

Workflow management systems are designed to assist groups of people in carrying out work procedures, and contain organizational knowledge of where work flows in the default case. A workflow management system is defined as "a system that helps organizations to specify, execute, monitor, and coordinate the flow of work items within a distributed office environment."[8]. The system is typically composed of two conceptual components: the specification module (or the "workflow model"), and the execution module (or the "workflow engine").

The first component, the specification module, enables administrators and analysts to define procedures and activities, analyze and simulate them, and assign them to people. This component can typically model control structures, data structures, goals, organizational structures, conversation structures, etc. Most workflow models capture (at least) procedures and the steps (or activities) which make up the procedures and the precedence ordering between activities. It is important that this component be sophisticated enough to capture the important aspects of the organizational environment, and simple enough to be dynamically changed when necessary.

The second component, the execution module, consists of the execution interface seen by end users and the execution environment which assists in coordinating and performing the procedures and activities. It enables the units of work to flow from one user's workstation to another as the steps of a procedure are completed. Some of these steps may be executed in parallel; some executed automatically by the computer system. The execution interface is utilized for all manual steps, and typically presents forms on the electronic desktop of appropriate users. The user fills in forms, or somehow executes a work step with the assistance of the computer system. Various databases, tools, and servers may be accessed in a programmed or ad hoc fashion during the processing of a work step.

A common belief has emerged in the workflow community that the specification and execution modules need to be tightly interwoven. For example, it should be possible to edit the workflow model of a procedure and thereby dynamically and safely change how the steps of the procedure are being executed. This belief is based upon the observation that change is way of life in most organizations. Those organizations which refuse to change are headed toward obsolescence because they cannot compete. Organizations must frequently make changes such as consolidating systems and procedures, improving business processes, and downsizing.

Changes frequently dictate other concomitant changes, so it is often necessary to do a set of changes as a unit. This can get quite complex and error prone. We thus see that the total change itself can

be viewed as a process containing some partial of activities. In practice many organizations find it necessary to suspend or abort work in progress in order to avoid undesirable side effects of complex change. This is often an inefficient, and ineffective solution because many organizations find it very unproductive, and sometimes impossible, to shut down all activities in order to make a change. From pharmaceutical factories to software engineering houses, this is a nagging problem - the bigger the organization, the more painful the change process. Today, large organizations usually do not solve this dynamic change problem in general, they cope, evade, or "muddle through." This document presents a set of concepts, constructs, and a language to help address this dynamic change problem.

In this document, we restrict our considerations to the issues of procedural change within workflow systems. The Collaboration Technology Research Group (CTRG) at the University of Colorado has been performing research in the groupware and workflow areas for the last eight years. Our prior work has considered issues of correctness and consistency of dynamic procedural change [9, 10] in workflow systems, and is complemented by the new concepts, insights, and language herein proposed. We are not concerned here with changes to the values of application data (which tends to be within the "routine" category of change), or other changes. The term dynamic means that we are making the change "on the fly" in the midst of continuous execution of the changing procedure. Dynamic procedural change is challenging, and sometimes produces "dynamic bugs" which can be very obscure and elusive in large workflows. Many of the techniques developed to handle these changes can be applied to other (non-procedural) changes.

A very simple example of dynamic procedural change which illustrates a dynamic bug is the following. An organization which traditionally does order processing performs the shipping step before the billing step. They make a dynamic change to their procedure to speed up the procedure by performing the shipping and billing steps concurrently. Although the procedure "looks safe" before the change, and "looks safe" for all orders begun after the change, there are problems (dynamic bugs) that could potentially surface during the change. For example, since orders that are in progress during the change are not flushed, some of these orders which are "inside" of the shipping step, will not perform the billing step at all, so some customers will not be billed. In this simple example, the problem may be intuitively obvious, but in realistically large workflows, the complexity is too high to catch all dynamic bugs via intuitive inspection. Thus, in the Ph.D. thesis of Keddara [14], and in other publications [9, 10], we have elucidated a formal model (timed Petri net extension), and a set of definitions and theorems which allow us to mathematically verify the correctness of certain classes of dynamic change, and to propose change structures which are error free.

During our years of work on the dynamic change problem, we have realized that there are numerous intricacies within the specification of a change that can be handled better if there is a concise and unambiguous change specification language. This is a language used by analysts or office administrators to specify one or more changes to an organizational procedure. The authors of this document have not found any change specification language in the literature; the change specification language presented in this document ($\mathcal{ML}$-$\mathcal{DEWS}$) is designed specifically with features for the complete and unambiguous description of procedural change. It requires, of course, a specification of the procedure before the change; and it can be used in conjunction with our correctness verification techniques. $\mathcal{ML}$-$\mathcal{DEWS}$, which abbreviates to Modeling Language to support the Dynamic Evolution of Workflow Systems, is intended for purposes of change understanding, change communication, change analyses, and change implementation. As presented in this document, $\mathcal{ML}$-$\mathcal{DEWS}$ specifications are not expected to be created by casual managers or end users. $\mathcal{ML}$-$\mathcal{DEWS}$ is an extensible language which is independent of the particular procedure to be changed, and independent of the particular workflow model and system used by any particular organization.

2 Related Work

There has been a large volume of work addressing workflow systems, but very little work concerned with the rigorous specification of dynamic change within workflows. There are many workflow products currently on the market [19], as well as research prototypes systems reported in the literature [15]. A lot of good work has been d one in the areas of workflow architectures, workflow models, and pragmatic workflow studies [13]. Although there have been workflow success stories, there have also been numerous cases in which workflow has not lived up to its expectations [24].

Traditional workflow systems (and office information systems before them) have been criticized in the literature as "automating a fiction" in the office because of their tendency to inflexibly prescribe temporal activity sequencing, and to narrowly dictate and restrict, rather than to broadly assist, in the roles that people play. People in the office typically engage in a lot of problem solving, informal communication, and exception handling [26]. In order to "get the work done" it may be necessary to creatively augment or circumvent standard office procedures. The mechanisms to help people do their necessary problem solving and decision making and exception handling are typically lacking in today's workflow products. Office work has been better characterized as "situated action" and "articulation work" [27], than its older description, derived from scientific management literature, as detailed procedure execution. Fortunately, we have seen more attention paid, in recent years to the above problems.

Another plaguing problem of workflow is the ever changing, ever evolving nature of the modern organization and its processes. Typical complex workflow management systems have not been able to fluidly evolve as is necessary in today's organizations. There have been indicators throughout the lifetime of the workflow literature of awareness of the problems of workflow evolution and migration. Some of the research beginnings in the workflow area come from one of the author's early work at Xerox PARC on OfficeTalk [11] and Information Control Nets [8]. Also, GMD Germany has continually had an interest in this area over the years beginning with Domino [16]. Usage reports of Domino detailed numerous problems and reasons for user rejection of the system including the ever changing nature of organizational processes. This problem of organizational process change is the motivator of this paper and related literature.

The first work in the literature which carefully motivated and mathematically articulated the issues of dynamic workflow change was a 1995 paper by the authors at the Organizational Computing Conference [9]. This paper began with justification for the investigation of dynamic change via examples of "dynamic bugs" which can yield surprising chaotic results if the change is done without care and precision. It presented a Petri net abstraction for modelling dynamic change. It showed that change to workflow procedures can be accomplished dynamically without shutting down the system, and without aborting or restarting work in progress. The paper rigorously defined the notion of dynamic change, and the notion of change correctness. The latter part of the paper introduced the SCOC transformation for dynamic change in which workcases can be gradually and efficiently migrated from the old procedure to the new procedure. One of the theorems in the paper established the correctness of SCOC. Recent followup work on this has included work by W.M.P. van der Aalst [28, 29], by Agostini and DeMichelis [1], and by Ellis, Keddara and Wainer [10]. Other highly related recent work includes the work reported in Klein[15], in Reichert[22], in Kumar[17] and in Casati[6].

In several reports by van der Aalst, he has used a Petri net model of workflow to derive mathematical techniques to verify the soundness property. A workflow process is sound if and only if, for any case, the process terminates properly, i.e., termination is guaranteed, there are no dangling references, and deadlock and livelock are absent.

The work of Agostini and DeMichelis addresses issues of dynamic process change using a simple restricted class of Petri net models named Free-Choice Acyclic Elementary Nets. They verify that change operations of parallelization, sequentialization, and swap can be performed safely within their

model. This work is integrated within Milano, a prototype CSCW platform developed within the Coordination Technology Laboratory at the University of Milano.

In their 1998 publication, Ellis, Keddara, and Wainer introduce timed flow nets as a model for analysis of workflow change which incorporates specification of temporal aspects and constraints. In this model, a time interval is associated with each step (or activity) of the procedure representing the minimum and maximum time that the activity might possibly consume. Thus, min-times and critical paths can be calculated for procedures, and various temporal conditions can be verified. This paper also expands upon the work in [9, 10] by analyzing and extablishing results concerning dynamic change composition and iteration. This work is part of an ongoing effort within the Collaboration Technology Research Group (CTRG) at the University of Colorado.

We have noted that dynamic change within a large workflow is complex, unintuitive, error-prone, and sometimes ambiguously specified. All of these issues strongly suggest that the change itself should be considered as a process, and that there is a strong need for a precise and concise Change Specification Language. The current paper thus offers a language, $\mathcal{ML}$-$\mathcal{DEWS}$, especially designed for specification of workflow change. It is a new and novel language. Note that $\mathcal{ML}$-$\mathcal{DEWS}$ assumes a process specification in some arbitrary language, as one of its given inputs. Thus, it is related to notions of standardization and codification of workflow languages because $\mathcal{ML}$-$\mathcal{DEWS}$ should ideally be able to work well with many different process languages. Thus work such as the Workflow Management Coalition [30] standardization efforts, and the Process Handbook [20, 18] are relevant.

The Workflow Management Coalition (WfMC) is a non profit organization representing numerous users, researchers, developers, and vendors of workflow management systems with the objective of advancing the opportunities for the exploitation of workflow technology through the development of common terminology and standards. It has been recognized that all workflow products have some common characteristics enabling them to potentially achieve a level of interoperability through the use of common standards for various functions. To this end, the WfMC has published documents of definition for common workflow terminology and interface standards [30]. This $\mathcal{ML}$-$\mathcal{DEWS}$ document relies heavily upon the terminology and definitions proposed by the WfMC.

The process handbook project is an effort to develop new methodologies for representing and codifying organizational processes at different levels of abstraction [20]. This is an ongoing effort in the Center for Coordination Science within the Sloan School at MIT. The related PIF project is to develop a Process Interchange Format (PIF) [18]. This is a form of process language that could serve as an intermediate form into which any workflow process description could be transferred. This would facilitate the sharing of process descriptions across diverse representations. PIF builds upon KIF (the Knowledge Interchange Format) developed for sharing knowledge bases, but extends it to provide process specification constructs. The hope is to evolve a notation and tool set for easily sharing descriptions of processes among many groups, and many computation tools.

As a final note concerning related work, we emphasize that dynamic process change is an important issue within numerous other domains such as software engineering [21], telecommunications service delivery [2], and flexible manufacturing [25]. We believe that the language articulated in this document can be immediately applied to these diverse other areas.

3 Modalities of Change

When one specifies a change, there are many factors which must be taken into account. Many of these factors can be considered as pre-conditions and post-conditions for change to happen. Within our model and language for change specification, pre-conditions are logical formula which must be fulfilled before the change can begin; post-conditions are logical formula which must be fulfilled in order for the change to be considered complete. These pre- and post-conditions may be expressed as functions

of time, application data, organizational context, process data, history, personnel data, state of the workcase, state of the total system, resource availability, and other exogenous information. The following is a typical change example.

Example 1. *Notice to all employees - Effective at the beginning of the work day on January 1, 2000: The Widgets Company will do shipping and billing in parallel (concurrently.) Also, a new computer system has been installed. Manual shipping and billing is being replaced by computerized. The new shipping and billing software will be up and working soon. All employees of shipping and billing should try out the new software on a few of your workcases within the next two weeks. You must be totally switched over to the computerized system by January 1, 2000.*

Note: A workcase must either be entirely processed by the computerized system, or entirely manually.

Exception: As you know, we occasionally have Saturday rush jobs which we always ship the same day, although billing does not work Saturdays. Thus billing for these must be done on the Monday after. On July 1, 2000, the billing department will work on Saturdays, so this exception terminates at that time.

As previously mentioned, a procedure may have a large number of workcases in progress at any given time. When a change is specified, it is also necessary to specify the subset of the workcases to which the change is applicable. In $\mathcal{ML}$-$\mathcal{DEWS}$, the specification of this subset is done via filters. For example, a change of top management approval may be instituted for all workcases involving more than 10,000 dollars. In general, subset selection may be a complex function. The $\mathcal{ML}$-$\mathcal{DEWS}$ change model and language also incorporates the notion of filters, similar in syntax to conditions, to select the workcases to which a change is applicable.

The temporal aspect of change is sometimes critical. In the above change example, the concept of "change specification rollout-time" denotes the date and time when the change is officially announced. This time specification acts as an anchor for other times which can be specified in absolute time, or in relative time. Relative is always with respect to the change specification time. Thus, we may have a change ramp up time , and a change cut-off time as specified in the above example. Note that in the example, the parallelism change specification roll-out time an absolute time is beginning of work day on January 1, 2000, whereas the beginning of automated shipping/billing is relative to the specification time (within 2 weeks.)

We next identify and explain five modalities of change that are important elements in any change specification. These modalities are duration, lifetime, medium, time-frame, and continuity. The lack of specification of these elements frequently leads to ambiguity - the manager distributes a statement of change, and the employees mis-interpret the statement, and the change is mis-implemented.

M1. Duration: Instantaneous VS time interval VS indefinite

One factor is the specification of whether the change is to happen quickly (instantaneously) or over a noticeably long (but finite and well specified) time period or an unspecified amount of time ("as long as it takes for the old stuff to change"). Frequently change that is immediate and instantaneous is desired by management, but sometimes an indefinite time period for change is preferred. An example of the latter (ongoing, indefinite time) is a change by introducing a new version of a software package. Some customers will immediately switch to the new version, and others may switch at a later time. Frequently there is an expressed commitment to maintain the old version for as long as customers are using the old version. Thus, the amount of time for all customers to switch from the old to the new version may be indefinitely long.

M2. Lifetime: Permanent VS temporary

Another factor is the amount of time that the change is in effect (with respect to the change specification roll-out time). If this time is specified as forever, then the lifetime is permanent. However, many changes are put in place for a specific and finite period of time. For example, a new set of procedures may be in effect for the next two week while the head manager is away on vacation only. Of course the nature of the temporary lifetime can be conditioned upon many factors such as customer satisfaction or time of new employee hire.

We believe that the notion of exception handling can fruitfully be considered as a special case of dynamic change whose lifetime is temporary, and whose filters may select one and only one workcase. Thus if it is decided that one particular customer must skip the time consuming credit approval activity, then we make a dynamic change to the procedure by omitting the credit approval step. This change is not permanent, but temporary; and we specify a filter which enables the change to be applicable only to this one customer.

M3. Medium: Manual VS automatic VS mixture

Most changes require different data and / or routing to occur for some number of customers (or workcases). When the number of workcases that must change is small, it is frequently done by a human who uses a medium such as pen and paper to make changes (and perhaps explanatory notes.) On the other hand, if there are thousands or millions of workcases which are effected by the change, then a computer program is typically written to allow the workcases which fit within the filter to be automatically updated. There are other media which have been used for change, and the media possibilities will continue to grow in the future.

M4. Time-frame: Past VS present VS future

In considering the workcases to which a change is applicable, one typically restricts consideration to workcases which are currently in progress (where current typically refers to the change specification roll-out time). This is an aspect in which ordinary English language specifications of change are sometimes unclear. It must be remembered that there are situations in which one must specifically exclude workcases which have not yet begun, or specifically include workcases which have already terminated. Thus we find change notifications which are retroactively applicable to old workcases - e.g.: "this ruling applies to all jobs completed in 1998 and after". This type of change may require that certain old workcases be updated. The filters within our change specification language are the mechanism to specify time-frame.

M5. Continuity: Preemptive VS Integrative

Every change requires some planning and some implementation work. Every change thus embodies a migration strategy. In the case of exception handling, the planning may necessarily be short in duration, and it may be highly intertwined with implementation at workflow enactment time. Never the less, we always must decide the various modalities, including whether we will somehow disrupt (or preempt) currently running workcases, or whether we will somehow allow the current workcases to continue for some time in a smooth fashion. Preemptive strategies include abort schemes, rollback schemes, restart schemes, checkpoint schemes, and flush schemes. Non-preemptive strategies, which we define as integrative strategies, include versioning, SCOC [9], and other gradual workcase migration schemes. In general, the specific requirements and desires and capabilities of an organization, and of a specific change dictate the choice between preemptive versus integrative continuity.

M6. Change Agents

This is a specification of which participants play which organizational roles within the change process. For example, it specifies who has the right to specify, enact, and authorize what types of changes. This is an important vehicle for *participatory change*. Further discussion of this modality is beyond the scope of this document.

4 The $\mathcal{ML}$-$\mathcal{DEWS}$ Workflow Model

$\mathcal{ML}$-$\mathcal{DEWS}$ is a modeling language which describes workflow changes. In order to specify a change, there must be a pre-existing description of the workflow before the change. This is the underlying workflow specification which is described in some modeling language such as IBM Flowmark[12], ICNs [8] or Action Workflows. Thus, $\mathcal{ML}$-$\mathcal{DEWS}$ is destined to lay on top of an underlying workflow model. In this section, we present a *basic* workflow model which is in accordance with the WorkFlow Management Coalition proposals [30] (the class diagrams are shown in the appendix.) Our claim is that $\mathcal{ML}$-$\mathcal{DEWS}$ can be easily adapted to any existing workflow model, notwithstanding that it *uses* this model.

4.1 The $\mathcal{ML}$-$\mathcal{DEWS}$ Definition Model

The WFMC reference model defines a *business process* as "a procedure where documents, information of tasks are passed between participants according to a defined sets of rules to achieve, or contribute to, an overall business goal" [30]. A *workflow* is a representation of the business process in a machine readable manner. A workflow model defines the different steps of the process execution, the flow of control and data among these different steps.

An activity definition specifies a logical step within a process. It has :

- a *name* which uniquely identifies the activity within a process.
- a *label* which is not necessarily unique. Thus, two activities may have the same label.
- a *category*; an activity can be *manual* or *automated*.
- a *type*; an activity can be be an *elemental* activity or a *macro* activity. A macro activity is *realized* by another process, referred to as the *realizer* of the activity, which is executed when the activity is started.
- a set of *pre-conditions*, which define when the execution of the activity starts.
- a set of *post-conditions*, which define when the execution of the activity completes.
- a set of *participants* who are eligible to participate in the activity.
- a *container*; a reference to the definition of the process which contains the activity.

A process definition specifies a process in a form which can be understood and enacted by a workflow management system. It has:

- a *name* which is not necessarily unique.
- a *version number* which changes as the definition of the process evolves. The combination name + version is unique.
- a possibly empty set of *pre-conditions* which specify when the execution of the process starts.
- a possibly empty set of *post-conditions* which specify when the execution of the process is to be considered completed.
- a *category*; A process can be either *structured*, *ad-hoc* or *hybrid*. If the activities, the sub-processes and their flow are fully defined in advance, then the process is said to be *structured*. If they are not defined in advance, then the process is said to be ad-hoc. Otherwise, it is said to be *hybrid*.
- a possibly empty set of activity definitions which define the logical steps of the process.
- a *flow* which specifies the order in which these activities are executed (*control flow*) and how the data is exchanged between these activities (*data flow*.) The activities are *connected* to each other using (data and flow) *connectors*.
- a set of *participants* who are eligible to participate in the process.
- a *form*; a reference to its application data definition.
- a possibly empty set of *constraints* which define the properties that the process executions ought to meet in the absence of any change.

The pre-conditions, the post-conditions and the constraints may be formulated based on two different types of information; *data* and *events* (sometimes called triggers). There are two kinds of data; *application data*, which represents the information being processed by the process, and *process data*, which provides information on the execution of the process; for instance whether an activity has successfully completed. There are also at least two kinds of events; *external events* which are externally *triggered* such as a *message* (be it an e-mail, fax, phone call or an EDI transaction), and *internal events* which are generated by the execution of the process itself.

Application data define the necessary information that a process execution manipulates; such information can affect the flow of control of a process. In the context of $\mathcal{ML}$-$\mathcal{DEWS}$, the application data are specified by a *form definition*. Each form definition has:

- a *name* which uniquely identifies the form.
- and a set of *field declarations*. A field declaration specifies the name of a field, its type, and optionally, a default value and an access control list (abbreviated to acl). The acl of a data field defines *how* the field can be accessed (e.g. Read/Write) and by *whom* (e.g. participants, activities, sub-processes.)

Traditionally, the organizational structure and its embedding into processes is reflected through the so-called *roles* and *actors*; actors *perform* roles, and roles are *assigned* to activities or processes. These perform/assign associations can be static (fixed), dynamic (e.g. rule-based), or hybrid. In order to hide these details, we uses the notion of *participant*[1] to reflect the involvement of actors in the execution of processes. Each participant has:

- a *qualified name*. This name is of the form '*actor_name::role_name::part_name*'. The part name defines the workflow part the actor is contributing to: for example, an actor a *performer*, a *responsible*, a *customer*, a *negotiator*, an *observer*, and others.

4.2 The $\mathcal{ML}$-$\mathcal{DEWS}$ Enactment Model

A process instance[2] represents a single[3] enactment of a process. There may be several instances associated with the same process, and simultaneously in progress within a workflow management system. However, we assume that these processes instances do not interfere with each other.

An activity instance represents an activity within a process instance, referred to as its *container*. The (simultaneous or otherwise) enactments of the activity during the lifetime of its container; are each represented by a single work item.

A work item represents a *concrete generic piece of work* to be handled by a group of participants. A work unit is associated with a single activity instance, referred to as its *container*, however an activity instance may generate one or more work items; each of which represents a separate enactment of the activity. An item is *assigned* to one or several *participants* via the so-called *in-basket*. We make no assumptions about the mechanism used by the underlying workflow management system to realize this assignment (i.e push vs pull), however we assume that it provides for *coordination* amongst the participants involved in the same work item.

When a process is used in a modular design, the process is instantiated whenever the activity it realizes is enacted. In this case, the work item associated with this enactment is referred to as the *container* of the process instance.

A workcase represents the dynamic aspect of a process instance showing the state of execution. It is a locus of control similar to a tokens in Petri nets.

[1] this notion is not restricted to humans

[2] process instance and case will be used interchangeably

[3] this requirement is also known as the *copy rule*.

A workflow management system maintains status and history information which relates the progress of a workcase toward achieving its business goals. This information may be used for monitoring, auditing, or synchronization, and includes among other things what it is commonly known as enactment *states*. In this paper, we use the basic set of states recommended by the WFMC.

In the context of this work, the basic structure of any enactment object (i.e. an activity instance, or a process instance, or a work item) consists of:

- a *name* which uniquely identifies the object.
- a *label* which is not necessarily unique.
- a *schema name*; the name of its definition (or the definition of its container, for a work item.)
- a *version number*[4]; the process version number attached to the object.
- an *open time*; the time at which the object was created.
- a *close time*; the time at which the object is closed (i.e. completed, terminated or aborted.)
- a *state*; the current state of the object.
- a *priority*; the priority of the object (if applicable.)
- a *form*; the application data processed by the object.
- a list of *participants* in the object execution. We assume that each enactment object has a participant responsible.
- a *container*; the reference to the object's container.
- a *root*; a reference to the process instance it is part of.

The application data processed by an enactment object is referred to as a *form instance*. Each form instance has:

- a *name* which uniquely identifies the form instance.
- a *label* which is not necessarily unique.
- a *schema* (i.e. its definition.)
- and a set of *data fields* given as a list of name/value pairs.

An *event* represents the occurrence of a special condition, be it internal or external, to which a workflow system may react by taking a specific course of actions. For instance, an event may be triggered when an application data changes, or the state of an object changes, or a participant assignment change occurs, or a message arrives, or a timer expires. Each event has:

- a *name* which uniquely identifies the event.
- a *type*, to identify the event type (e.g. data change, state change etc...); a basic list may be found at the end of the paper.
- a *producer*, to identify the object which has caused the event (if applicable.)
- a *root*, typically the process instance associated with the producer (if applicable.)
- an *event time* which reflects the time at which the event has occurred.
- and some *event data* which depends upon the event type.

Execution audit data is maintained, for each process instance, in an execution history. Each execution history has:

- an *owner*; the process instance.
- the list of closed work items.

[4] this information may be necessary in the context of process changes

5 The $\mathcal{ML}$-$\mathcal{DEWS}$ Change Model

In the context of $\mathcal{ML}$-$\mathcal{DEWS}$, the emphasis is put on process changes, other types of change such as change of organizational structures, change of social structures, albeit important, are beyond the scope of this work. In the sequel, we shall use change to speak of process change. We will also use interchangeably *schema* and definition.

A change has two facets; namely *schema changes* and *instance changes*. A schema change occurs when the definition of a business process is modified. An instance change occurs when a process instance changes; for example an exception represents a form of instance change; it occurs when a process execution deviates from its definition as the result of an enactment error (e.g. constraints violations) or an unexpected situation (e.g. a workers strike.)

The underlying philosophy of our *unified* change model is based on three key observations:

- Change specification is a, albeit complex, process specification which describes the steps and their sequencing, the participants involved in the change, the change data and the change constraints.
- schema changes, in general, yields to some of form of instance changes.
- Instance changes, of which exceptions are special cases, frequently, may be assimilated to *temporary* schema changes.

In the context of this work, we focus on change specification. The authors have dealt with change analysis in previous published papers[9, 10]. Change enactment will be dealt with in a forthcoming paper.

A change means that a process definition, referred to as the old version of the underlying process, is transformed into a new process definition, referred to as the new version of the underlying process. Such a change, frequently, requires the cases which represent executions of the old version to *migrate* to the new version. A more global view is to consider such a change as one ring in a chain of changes, and as such, the change may also affect the cases which are migrating to, or have migrated to the old version. To accommodate this behavior, a case should no longer be *associated* with a single process definition, but possibly with several distinct process definitions. Each process definition represents a version in a previously carried out change. In the context of a change, an old case represents a case which is *associated* with the old version. The change transforms the old case into a new case in accordance with the underlying change enactment. Although, the description of the enactment model is beyond the scope of this work, we assume that the notion of case migration is supported.

A change definition specifies a change in a form which can be understood and used by all parties involved in a change, to communicate, and carry out their responsibilities. Each change definition is an object which consists of:

- a *name* which uniquely identifies the change.
- a *description* which may identify the *goals*, *the motivations*, and the *assumptions* behind the change.
- a set of *change participants* which identifies the parties and their roles in the change; for example *change agents* are the participants who craft the change specification. We assume that a change has a participant responsible.
- a *specification roll-out time* which defines the time at which the change is officially announced.
- the *old version* and the *new version* of the underlying process.
- a set of *global rules* which apply to a new case, regardless of the policy it *uses* as a basis for migration.
- a set of *change policies* which address the instance change issues as explained next.

A change policy definition *completes* the change specification; by addressing the *which-how-when* questions related to instance changes. Each change policy definition is an object which consists of:

- a name which uniquely identifies the change policy within a change.
- a reference to the change, the policy is associated with.
- a set of *filters* to identify the old cases the policy applies to.
- a *scheme*; our model supports a variety of pre-defined instance change schemes; including *SCOC* (versioning), *E-SCOC* (delayed migration), *Abort*, *Resubmit*, *Flush* and *Roll Back*.
- a *mode* to define whether the change is to be carried out in a *manual*, *automated* or *hybrid* fashion.
- a set of *local rules* that apply specifically to any new case which *uses* this policy as a basis for migration to the new version.
- a *change process* which describes the activities and their sequencing, and the participants involved in the change (to be explained shortly.)

In the context of $\mathcal{ML}$-$\mathcal{DEWS}$, a filter is a rule which selects a subset of cases for change. It may be formulated based on a variety of factors; including:

- the state information, the timing information, application data, participants, execution history of the old case.
- The state, timing, history information on any change, previously carried out upon the old case.

Each rule has a *name*, an (optional) modal-part, a guard-part, an action-part, and an (optional) exception-part. The modal part determines when the *guard-part* is evaluated. It may be either a temporal or an event property. The guard part is a logical formula which is evaluated. If it is satisfied, then the action part is executed, else the exception part is executed (for more information on rules, the reader is referred to the next section.)

Filtering varies from one change to another. For example, in some situations, it may be desirable to *filter-in* the old cases which have already completed (e.g. recalling defected parts.) In other situations, it may be necessary to *filter-out* the running old cases upon which a previous change has been enacted (e.g. irreconcilable changes.) Traditionally, only the running cases over the old version have been considered by the different migration models proposed in the emerging literature; this work breaks away from this tradition.

Example 2. A scoping statement such as "*all the orders which have started after 10:00 a.m., have not yet been shipped , and have a value of more than 1000 dollars*", may be expressed in $\mathcal{ML}$-$\mathcal{DEWS}$ as follows:

$$
\begin{array}{l}
[\underline{Rule}\ \ filter_1 \\
\quad [\underline{If}\ \ ((oldCase.state = \underline{Running})\ \ \underline{and}\ \ (oldCase.form('balance') > 1000) \\
\qquad\qquad \underline{and}\ \ (oldCase.state('Ship') \mathrel{!}= \underline{Completed}) \\
\qquad\qquad\quad \underline{and}\ \ (oldCase.start\_time \geq' 10:00\ a.m.'))\ ] \\
\quad [\underline{Then}\ \ \underline{filterIn}(oldCase)] \\
\quad [\underline{Else}\ \ \underline{filterOut}(oldCase)] \\
]
\end{array}
$$

Here, *oldCase* is a *free variable* which is to be bound by the system to an old case. $oldCase.state('Ship')$ returns the state of the shipping activity within the old case. $\underline{filterIn}(oldCase)$ and $\underline{filterOut}(oldCase)$ instruct the workflow system to filter-in or filter-out the old case.

Sometime, a filtering statement may be a viewed as a dynamic property whose truth value may vary with time. This means that an order may be filtered out one moment, and filtered in the next moment. To this end, $\mathcal{ML}$-$\mathcal{DEWS}$ provides modal clauses to deal with such situations.

Example 3. Consider a filtering statement which reads "*An order is eligible to transit to the new system if the participant responsible for the order gets notified*". This statement may be expressed in $\mathcal{ML}$-$\mathcal{DEWS}$ as follows:

$$
\begin{array}{l}
[\underline{Rule}\ \ filter_2 \\
\quad \langle \underline{When}\ \ (\underline{Event}\ \underline{Notified}\ \ oldCase\ \text{'switch to new procedure'}) \rangle \\
\quad [\underline{If}\ \ (oldCase.state = \underline{Running})] \\
\quad [\underline{Then}\ \ \underline{filterIn}(oldCase)] \\
\quad [\underline{Else}\ \ \underline{filterOut}(oldCase)] \\
]
\end{array}
$$

The modal part of this rule, specified by the *When*-clause, is satisfied when the notification event, is triggered. When this happens, a case is filtered-in if it is running, otherwise it is filtered out. For each old case, this rule fires when the event is triggered.

The local and the global change constraints, formulated by rules, express various properties that a new case ought to fulfill in the course of its lifetime; including temporal properties. Time, in general, can be an ambiguous notion, and expressing temporal properties may be a confusing process. One difficulty stems from the variety of *special* moments in the lifetime of a case migration (note here that special is a relative term and this is precisely the point.) Thus, pre-defining these special moments is not an easy task, to say the least.

$\mathcal{ML}$-$\mathcal{DEWS}$ provides some relief by offering the ability to attach "on the fly" variables, referred to as slots, to a new case. These may be either instance slots (the default;) that is each new case has its own copy of the slot, or class slots; that is one copy is shared by all new cases. In general, a dynamic slot is related to (external or internal) events, and thus its value may be set by a rule. For example, the following $\mathcal{ML}$-$\mathcal{DEWS}$ statements add a dynamic instance slot *sb_time* to the new case, and define two rules which, each time either the shipping or the billing activity is completed, set *sbtime* to the corresponding completion time (here, *newCase* is a free variable which is bound by the system to a new case:)

$$[\underline{addslot}\ \ newCase\ \ sbtime\ \ \underline{Time}\ \underline{Instance}\ ]$$

$$
\begin{array}{l}
[\underline{Rule}\ \ rule_1 \\
\quad \langle When\ \ evt\ \ (\underline{Event}\ \ \underline{Completed}\ \ newCase\ \ 'Shipping') \rangle \\
\quad [\underline{If}\ \ \underline{True}] \\
\quad [\underline{Then}\ \ (newCase.sbtime\ \ \underline{is}\ \ evt.event\_time)] \\
]
\end{array}
$$

$$
\begin{array}{l}
[\underline{Rule}\ \ rule_2 \\
\quad \langle \underline{When}\ \ evt\ \ (\underline{Event}\ \ \underline{Completed}\ \ newCase\ \ 'Billing') \rangle \\
\quad [\underline{If}\ \ \underline{True}] \\
\quad [\underline{Then}\ \ (newCase.sbtime\ \ \underline{is}\ \ evt.event\_time)] \\
]
\end{array}
$$

Example 4. A constraint which reads as "*the transition to the new version should start today and completes before the end of the day*", may be expressed as follows (assuming the the name of the change is 'Change:1.0')

$$
\begin{array}{l}
[\underline{addslot}\ \ newCase\ \ chg\_start\ \ \underline{Time}\ \underline{Instance}\ ] \\
[\underline{addslot}\ \ newCase\ \ chg\_end\ \ \underline{Time}\ \underline{Instance}]
\end{array}
$$

$$
\begin{array}{l}
[\underline{Rule}\ rule_3 \\
\quad \langle \underline{When}\ evt\ (\underline{Event}\ \underline{Started}\ newCase\ 'Change:1.0') \rangle \\
\quad [\underline{If}\ \underline{True}] \\
\quad [\underline{Then}(newCase.chg\_start\ \underline{is}\ evt.event\_time.day)] \\
] \\
[\underline{Rule}\ rule_4 \\
\quad \langle \underline{When}\ evt\ (\underline{Event}\ \underline{Completed}\ newCase\ 'Change:1.0') \rangle \\
\quad [\underline{If}\ \underline{True}] \\
\quad [\underline{Then}\ (newCase.chg\_end\ \underline{is}\ evt.event\_time.day)] \\
] \\
[\underline{Rule}\ rule_5 \\
\quad [\underline{If}\ ((newCase.chg\_start\ !=\ \underline{Today})\ \underline{or}\ (newCase.chg\_end\ !=\ \underline{Today}))] \\
\quad [\underline{Then}\ \underline{notify}(newCase.responsible)] \\
]
\end{array}
$$

Here, $\underline{Today}$ is a temporal sytem variable which is binded to the system's calendar day. The first rule states that whenever the event specified by the when-clause is triggered within the new case, in this case the migration of the new case to the new version starts, $time_1$ is set to the time at which the event is triggered. Likewise, the second rule states that whenever this migration ends, $time_2$ is set to the time at which the event is triggered. The third rule states that if the constraint is violated, then the participant responsible for the case is notified.

In some situation, a change constraint may also express a *collective* property that the new cases must meet, as illustrated by our next example.

Example 5. Consider the case of a constraint which stipulates that "*the number of cases simultaneously migrating to the new procedure should always be kept below 10.*

First, a dynamic class slot, *numcases*, which reflects the number of migrating instances is defined as follows:

$$[\underline{addslot}\ newCase\ numcases\ \underline{Integer}\ 0\ \underline{Class}\ ]$$

Next, two rules which modify the value of *numcases* are defined. The first rule states that whenever the change named 'Change:1.0' starts, *numcases* is incremented, and whenever it completes, *numcases* is decremented.

$$
\begin{array}{l}
[\underline{Rule}\ rule_6 \\
\quad \langle \underline{When}\ (\underline{Event}\ \underline{Started}\ newCase\ 'Change:1.0') \rangle \\
\quad [\underline{If}\ \underline{True}] \\
\quad [\underline{Then}\ (newCase.numcases\ \underline{is}\ (newCase.numCases + 1))] \\
] \\
[\underline{Rule}\ rule_7 \\
\quad \langle \underline{When}\ (\underline{Event}\ \underline{Completed}\ newCase\ 'Change:1.0') \rangle \\
\quad [\underline{If}\ \underline{True}] \\
\quad [\underline{Then}\ (newCase.numcases\ \underline{is}\ (newCase.numCases - 1))] \\
]
\end{array}
$$

Finally, the constraint is expressed by the following rule:

$$
\begin{array}{l}
[\underline{Rule}\ rule_8 \\
\quad [\underline{If}\ (newCase.numcases > 10)] \\
\quad [\underline{Then}\ \underline{notify}(newCase.responsible)] \\
]
\end{array}
$$

Yet, in other situations, a change constraint may not express migration properties *per say*, but it is used to *adjust* execution constraints which can no longer be met due to the change; e.g. scheduling or trace constraints. This point is illustrated in the following example.

Example 6. Consider a change statement excerpt which reads "*...because the newly introduced testing procedure is crucial to the yield of our manufacturing process, any disk drive in transition to the new process must undergo the new testing, even if it has already been tested by the old testing procedure. Therefore, the expected delivery time of 8 hours may be extended to 12 hours for such a hard drive*". This statement may be expressed in $\mathcal{ML}$-$\mathcal{DEWS}$ using three constraints. The first constraint says that when a disk drive is delivered, its case history contains the activity name 'New Test'. The second constraint says that if the hard drive has been tested by the activity name 'Old Test', then the delivery time (i.e. his case time duration) may not exceed 12 hours. Otherwise, as stated by the three rules, the delivery time should not exceed 8 hours.

$$
\begin{array}{l}
[\underline{Predicate}\ rule_7 \\
\quad \langle \underline{When}\ evt\ (\underline{Completed}\ newCase) \rangle \\
\quad [\underline{If}\ newCase.\underline{trace}('New\ Test') = 0)] \\
\quad [\underline{Then}\ \underline{activate}('New\ Test')] \\
]
\end{array}
$$

$$
\begin{array}{l}
[\underline{Predicate}\ rule_8 \\
\quad [\underline{If}\ ((newCase.\underline{trace}('Old\ Test') = 1)\ \underline{and} \\
\qquad (newCase.close\_time \geq (newCase.open\_time +' 12:00')))] \\
\quad [\underline{Then}\ \underline{notify}(newCase.responsible)] \\
]
\end{array}
$$

$$
\begin{array}{l}
[\underline{Predicate}\ rule_9 \\
\quad [\underline{If}\ ((newCase.\underline{trace}('Old\ Test') = 0)\ \underline{and} \\
\qquad (newCase.close\_time \geq (newCase.open\_time +' 8:00')))] \\
\quad [\underline{Then}\ \underline{notify}(newCase.responsible)] \\
]
\end{array}
$$

Here, $\underline{activate}('New\ Test')$ instructs the system to activate the new test procedure. $newCase.\underline{trace}(\underline{name} : \underline{String})$ is a method which returns 1 if the execution trace (i.e. the sequence of the completed activities) of $newCase$ contains the named activity, 0 otherwise.

The change process specifies how a change is carried out. It is a process (w.r.t. our model;) the process activities are referred to herein as the *change activities*, the process participants are referred to as the *change participants*, and the flow is referred to as the *change flow*.

Change activities may be classified as meta or standard activities. A change meta-activity includes one or more actions which represent un-interruptible operations of finite duration to be performed *on* the new case: for example, changing the state, the data , or the participants of the case. The standard activities reflect what is generally perceived to be an activity. A change activity may be either manual or automated, elemental or a macro, etc... In one word, it is a full-fledged activity in accordance with our model. The specification of a change meta-activity include one or more actions which, for the purpose of this work, may change the state, the data, the participants of the new case, or:

- *move* a work item from the old version to the new version.
- *flush* a work item of the old version.
- *inject* a work item into the new version.

- *bridge* the old version to the the new version; this is a high level operation, the purpose of which, is to allow work items to move from the old to the new in an "asynchronous way" (i.e. without synchronized calls to the previous operations.) A bridge has a set of *in-lets* which are plugged into connectors in the old version, and a set of *out-lets* which are plugged into connectors in the new version. This dynamic bridging opens uncharted waters in instance migration; for instance collaborative migration as described shortly can be supported.

  All the migration strategies discussed in the literature can be supported through jumpers; including Synthetic Cut-Over Change (abbreviated to SCOC), Extended Synthetic Cut-Over Change (abbreviated E-SCOC), Resubmit, Wait, Flush, Transfer, Progressive. It also provides for the following new migration (extension) strategies:

  1. *corrective migration*; where by corrective activities, which are part of neither the old version, nor the new version, are performed during the case migration.
  2. *collaborative migration*; where by special kinds of change meta activities, referred to as *collaborative*, may be included in the change process. A collaborative allows participants to decide on the migration of some of this work, interactively. After the activity is completed, a bridge may be either manually or automatically established, based on the outcome of the collaborative.

  The reader is referred to the case study for an example of change process.

6 The $\mathcal{ML}$-$\mathcal{DEWS}$ Change Language

$\mathcal{ML}$-$\mathcal{DEWS}$ is an extensible language geared to the specification of workflow change and change policy. It is extensible in the sense that it provides a modeler with the ability to extend the basic workflow model through *sub-classing*, or to load her own model class definition.

Its declarative aspect allows a user to define predicates and combine them into complex expressions using the logical connectors *and*, *or* and *not*. A user may also define rules which manipulate objects. The modal operators of the language provides the ability to define dynamic rules whose interpretation may vary with time. This feature is well suited to accommodate the dynamic nature of workflow processes and their evolution. Its procedural aspect is based on the Java programming language.

6.1 Overview

In this section, we provide an overview of the declarative aspect of the language (a more detailed language description is available as a technical report.) More specifically, we describe the declarative aspects of the language.

Slots

An object has attributes, referred to herein as slots, and methods. As we saw earlier, a dynamic slot may be attached to a new case or shared with other new cases. In the former case, the slot is called an instance slot, and in the latter case, it is called a class slot. Unless a slot is initialized, its value is undefined. In this case, the evaluation of any logical formula results which uses the slot is undefined. For example, a condition such as

$$case.close\_time \leq' 8:00p.m.$$

is undefined if the case is still running.

Event & Time Specification

Event and time may be used in rules, predicates to express modal properties; i.e. properties whose interpretation changes as events are triggered or as time progresses. The basic event specification has the following syntax:

$$'(' \ \underline{Event} \text{ event-code case-exp [prod-exp] [event-data] } ')'$$

where

- event-code defines the event type and depends upon the underlying enactment model.
- case-exp defines the case within which the event occurred; it can be either the name of the case, or a variable which is bound to a case or a case name.
- prod-exp defines the producer which has produced the event, it can be either the name of the producer, or a variable which is bound to a producer or a producer name. A producer may be an activity, a process, a work item, or a change.
- event-data defines the event data, and depends upon the event type.

For example, the specification,

$$(\underline{Event} \ \underline{Completed} \ 'Order : 1.0' \ 'Shipping')$$

represents the event triggered when the activity named 'Shipping' within the process named 'Order:1.0' is completed. The specification,

$$(\underline{Event} \ \underline{Changed} \ 'Order : 1.0' \ (\text{ 'Reservation Status' 'Checked In' 'Checked Out' }))$$

represents the event triggered when the reservation status of the process named 'Order:1.0' is changed from 'Check In' to 'Checkout '. $\underline{Any}$ is a wild card which may be used as an event-code, case-exp, prod-exp.

Temporal properties may be defined using temporal operators (to be defined next.) A time specification may define time, date, or both. It may be a temporal user-defined variable, a system variable such as $\underline{Today}$, a string literal such as '01/01/1999 08:00 a.m.', an expression such as $\underline{Now} +' 2 : 00'$ (where *chg_time* is a temporal variable.)

Modal Clauses

A modal clause may be specified in a rule to signify when the guard of the rule is evaluated. A modal clause consists of a *modal operator* followed by an event or a time specification. $\mathcal{ML}$-$\mathcal{DEWS}$ provides the following syntax to define modal clauses:

$$'\langle' \text{ modal-operator } [variable] \text{ event-spec } | \ time - spec \ '\rangle'$$

where

- modal-operator is either $\underline{When}$ for an event specification, $\underline{At}$, $\underline{After}$, or $\underline{Before}$ for a time specification.
- variable, if specified, is bound to the specified event or time; this variable may be subsequently used in the rest of the rule.

The *When*-clause is satisfied only when the specified event occurs. The $\underline{At}$-clause is satisfied at the momeent defined by the time specification. The $\underline{Before}$-clause is satisfied any moment prior to the sepcified time. The $\underline{After}$-clause is satisfied any moment after the specified time.

Predicates

Predicates may be built and combined into complex logical formula using the logical connectors *and*, *or* and *not*, the existential quantifier *exist*, and the universal quantifier *forall*. A predicate has a *header* which consists of the name of the predicate, and a possibly empty list of parameter definitions. Each parameter represents a *free variable* which is bound when the predicate is *interpreted* (i.e. evaluated.) It has also a *guard*, a logical formula build using simple conditions on objects, relational and arithmetic expressions, user-defined predicates, all combined using the logical connectors and the quantifiers mentioned earlier. The general syntax of a predicate is:

$$'[' \ \underline{Predicate} \ \text{pred-name} \ '('[param-list]')' \ '[' \ \text{guard} \ ']' \ ']'$$

For instance, the following predicate may be used to see if the name of a participant in a given case, matches a pattern.

$$\begin{array}{l}[\underline{Predicate}\ \ Find\ \ (case : \underline{Case}, pattern : \underline{String}) \\ \quad [\underline{exist}\ \ part\ \ \underline{in}\ \ case.parts,\ \ (part.name\ \ \underline{matches}\ \ pattern)] \\ ]\end{array}$$

Rules

$\mathcal{ML}$-$\mathcal{DEWS}$ allows a user to define rules. Each rule has a *header* which consists of a *name* and a priority (0 by default). It has also an (optional) modal-part, a guard-part, an action-part, and an (optional) exception-part. The modal part determines when the *guard-part* is evaluated. It is a logical formula which is evaluated. If it is satisfied, then the action part is executed, else the exception part is executed. Rule selection is based first on priority, and then non-deterministically. The general syntax of a rule is:

$$\begin{array}{l}'[' \ \underline{Rule} \ \text{rule-name} \ \ [\text{priority}\,] \ [\ '\langle' \ \text{modal-part} '\rangle' \ ] \\ \qquad\qquad '[' \ \underline{If} \ \text{guard-part} \ ']' \ '[' \ \underline{Then} \ \text{action-part} \ ']' \ [\ '[' \ \underline{Else} \ \text{exception-part} \ ']' \ ] \\ ']'\end{array}$$

6.2 An Example Of Change Specification

In this section, we examine the statement of change formulated in example1, and illustrate how it can be expressed within $\mathcal{ML}$-$\mathcal{DEWS}$. For this example, we focus on the change rules; we do not show the procedures to be changed, nor do we describe the change processes. The case study will illustrate an example of change process.

This statement talks about two changes: the first changes *computerizes* the shipping and the billing activities, and the second change *parallelizes* the computerized versions of these activities.

The code below depicts the specification of the first change. This specification has a *header* which contains the name of the change, *Change* : 1.0, the specification roll-out time, '12/01/1998', the agent who crafted the specification, 'Karim Keddara:agent', the name of the old version, 'Process:1.0', the name of the new version, 'Process:2.0', and the name of the only change policy, 'Policy:1.0'.

The declarative code following the header, defines the dynamic slots and the global rules of the change. The slot *afew* is used to specify what the word 'few' means in the statement (here, it means at least 2.) The slot *av_time* is used to determine the time at which the new system is available for use. The slot *numcases* is used to keep track of the number of cases which have used the new system. The slot *chg1start_time*, and the slot *ch1end_time*, if defined, specify the time at which the migration of the new case starts and the time at which it ends.

$rule_1$ set the software availability time. Here, a user-defined event *SoftAvailable* is triggered when the software is available. $rule_2$ and $rule_3$ set the time at which the change starts and the time

at which the change ends. In $rule_4$, *numcases* is incremented each time the computerized billing activity is completed.

The change constraints are formulated by the $rule_5$, and $rule_6$: $rule_5$ notifies the case responsible if fewer than 2 employees use the new system within 2 weeks of the change specification rollout-time (represented here by the variable *rollout*.) $rule_6$ notifies the participant responsible for the case if the case uses the new system before it is available for use, or if the change does not end by year 2000.

This change has a single policy which filer-in only the cases which happen to be running when the change is rolled out. The policy stipulates that the transition is done manually, using SCOC. This means that the shipping and the billing are done either manually or using the new system.

```
{ChangeSpec
  Name 'Change:1.0' On '12/09/1998' By 'Karim Keddara:agent'
  OldProcess 'Process:1.0' NewProcess 'Process:2.0' Policies 'Policy:1.0'
  [
    [addslot newCase afew Integer 2 Class]
    [addslot newCase av_time Time Class ]
    [addslot newCase num_cases Integer 0 Class ]
    [addslot newCase chg1start_time Time ]
    [addslot newCase chg1end_time Time ]
    [Rule rule1
          ⟨When evt (Event SoftAvailable newCase)⟩
          [If True] [Then (newCase.av_time is evt.event_time)]
    ]
    [Rule rule2
          ⟨When evt (Event Started newCase 'Change : 1.0')⟩
          [If True] [Then(newCase.chg1start_time is evt.event_time)]
    ]
    [Rule rule3
          ⟨When evt (Event Completed newCase 'Change : 1.0')⟩
          [If True] [Then (newCase.chg1end_time is evt.event_time)]
    ]
    [Rule rule4
          ⟨When (Event Completed newCase 'Comp_Billing')⟩
          [If True] [Then(newCase.num_cases is (newCase.num_cases + 1))]
    ]
    [Rule rule5
          ⟨At (rollout +' 14 days')⟩
          [If (newCase.num_cases < newCase.afew)]
          [Then notify(newCase.responsible)]
    ]
    [Rule rule6
          [If ((newCase.av_time ≥ chg1start_time) or (chg1end_time ≥' 01/01/2000'))]
          [Then notify(newCase.responsible)]
    ]
  ]
  {PolicySpec
```

```
        Name 'Policy:1.0' Medium 'Manual' Scheme 'SCOC'
        [
            [Rule rule7
                    [If (case.state = Running)]
                    [Then filterIn(oldCase)]
                    [Else filterOut(oldCase)]
            ]
        ]
    }
}
```

The specification of the second change is defined below. This change has a single policy named 'Policy:1.0' which filter-in all the cases which are in progress after the new millenium ($rule_8$.) $rule_9$ and $rule_{10}$ set the change start time, *chg2start_time*, and the change end time, *chg2end_time*, of the new case. $rule_{11}$ is used to detect if the change is instantaneous.

To deal with the "saturday exception", we use two slots *shipping_day* and *billing_day* to keep track of the day at which the shipping and the billing activities are completed. These slots are set by $rule_{13}$ and $rule_{14}$. $rule_{12}$ notifies the case manager, if after july 1st, an order is not shipped and billed the same day it gets into the system. The assumption here is that prior to this change, this constraint was not maintained for the saturday orders, and that the change statement removes this exception to the rule.

```
{ChangeSpec
    Name 'Change:2.0' On '12/09/1998' By 'Karim Keddara'
    OldProcess 'Process:2.0' NewProcess 'Process:3.0' Policies 'Policy:1.0'
    {PolicySpec
        Name 'Policy:1.0' Medium 'Manual' Scheme 'ChangeProcess:0.1'
        [
            [addslot newCase chg2start_time Time ]
            [addslot newCase chg2end_time Time ]
            [addslot newCase billing_day Time ]
            [addslot newCase shipping_day Time ]
            [Rule rule8
                    ⟨At '01/01/2000 12:00 a.m.')⟩
                    [If (oldCase.state = Running)] [Then filterIn(oldCase)]
                    [Else filterOut(oldCase)]
            ]
            [Rule rule9
                    ⟨When evt (Event Started newCase 'Change : 2.0')⟩
                    [If True] [Then (newCase.chg2start_time is evt.event_time)]
            ]
            [Rule rule10
                    ⟨When evt (Event Completed newCase 'Change : 2.0')⟩
                    [If True] [Then (newCase.chg2end_time is evt.event_time)]
            ]
            [Rule rule11
                    [If (newCase.chg2end_time != newCase.chg2start_time)]
                    [Then notify(newCase.responsible)]
            ]
```

```
        [Rule rule12
                ⟨After '07/01/2000 12:00 a.m.'⟩
                [If ((newCase.shipping_day != newCase.billing_day)
                      or (newCase.shipping_day != newCase.start_time.day)) ]
                [Then notify(newCase.responsible)]
        ]
        [Rule rule13
                ⟨When evt (Event Completed newCase 'Comp_Bill')⟩
                [If (evt.event_time ≥' 07/01/2000 12:00 a.m.')]
                [Then (newCase.billing_day is evt.event_time.day)]
        ]
        [Rule rule14
                ⟨When evt (Event Completed newCase 'Comp_Ship')⟩
                [If (evt.event_time ≥' 07/01/2000 12:00 a.m.')]
                [Then (newCase.shipping_day is evt.event_time.day)]
        ]
    ]
  }
}
```

7 Case Study

For our case study, we have chosen a business process within a fictitious hotel chain called *The Desert Inn* (this story has been inspired from the running example of Bichler et al. [4].) This process handles many of the hotel activities including reservation, billing, check in and check out. The business model of *The Desert Inn* is based on *service differentiation*; there are two kinds of customers, namely *Highly Important Customers (HIC)s* and *Very Important Customers (VIC)s* (usually, everybody gets a high designation.)

7.1 The story

In a typical situation, a customer, either in person (or through a travel agent,) makes a reservation by calling the toll-free number 1-800-DESERT-IN of the hotel reception desk. A host greets the customer, collects the necessary information, and processes the customer request while the caller is waiting. The request is either rejected or confirmed. In the latter case, a confirmation number is issued to the customer. In some cases, the requests are processed off-line during the next business day.
A customer may at any time cancel a reservation; the *Desert Inn* waives the cancellation fee for *HIC*s, but usually charges *VIC*s the equivalent of one night stay. In the latter case, a bill is sent to the customer, a payment due reminder is set up, and if a payment is not received within one month, the case is handed over to the collection agency for legal actions.

A *VIC* pays the bill before checking out, whereas a *HIC* pays after receiving the bill.

The initial version of this process, referred to as *DI_Process:1.0*, is depicted in figure 3.

When a customer call is routed to a live agent, it is the beginning of a case. The activity *Greet Customer* routes the call to a receptionist who greets the customer, collects the necessary information, and flags the reservation as either *HIC* or *VIC*. In the former case, the request is processed by the activity *Online Processing* while the customer is on line; a confirmation number is issued to the

customer, and the reservation is flagged as confirmed (here, we assume that a *HIC* reservation is very seldom rejected.) In the latter case, the reservation is flagged as 'pending action', and it is queued in the *pending requests basket* to be processed during the next business day by the activity *Batch Processing.* If the reservation is rejected, then a rejection letter is fax-ed to the customer by the activity *Notify* , then the reservation is archived by the activity *Archive.* Otherwise, the reservation is flagged as confirmed, then a confirmation letter is fax-ed to the customer by the same activity *Notify* , and the case is queued waiting for either the customer to show up or to cancel.

When a *VIC* with a confirmed reservation *arrives* to the hotel, her check-in formalities are handled by the activity *Check-in* which flags the reservation as checked-in, and the account as pending billing. When her hotel stay comes to an end, she is billed and the account is flagged as 'Pending Payment (activity *Bill*). Then, she pays her bill and the account is flagged as payed (activity *Pay.*) Then, she must check out before 6.00 p.m. At the end of the activity *Check out*, the reservation is flagged as checked-out, then the reservation is archived as outlined earlier. The scenario is similar for a *HIC*, except for the fact that a *HIC* gets the hotel bill and pays for her bill after she checks out.

When a *VIC* with a reserved confirmation cancels her reservation (activity *Cancel,*) she is charged a fee, the reservation is flagged as canceled, and the account as pending payment (activity *Charge Fee.*) Then, a single reminder is set by the activity *Reminder.* If the payment is not received within one month, the account is flagged as delinquent, and the case is sent to the collection agency (activity *Collection.*)

To represent the reservation procedure, we use a model called *Information Control Nets* (abbreviated to ICN.) In its basic structure, an ICN is made of nodes and edges which connect the nodes. They are two types of nodes; *activity nodes*, represented as ellipses, and *control nodes*, represented as circles. An activity node represents an activity, it has a single (control) input node which signals the beginning of the activity, a single (control) output node which signals the end of the activity, and it is labeled with the activity label (note here that two nodes may have the same label.)

A control node is used for the control flow. Each process has a single *start control node* which signals the beginning of the execution of the process, and a single *end control node* which signals the end of the execution of the process. These nodes are easily distinguishable; the start control node has no incoming edge, and the end control node has no outgoing edge. *And-nodes*, represented by solid circles, are used to specify parallelism: for example or-fork and or-join primitives may be specified. *Or-nodes*, represented by hollow circles, are used to implement choice and non-determinism; a test may be attached to an or-node, and the outgoing edges may be labeled with the possible outcomes of the test. For example in the ICN representing DI_Process:1.0, the test associated with p_1 determines whether the customer status is a *VIC* or an *HIC.* The test associated with p_3 is used to determine if the reservation is rejected or confirmed. Yet another test is associated with p_{13} determine if the delay set by the activity *Charge Fee* has expired or not. In this example, we have no and-nodes.

The execution of an ICN starts when a *token* is deposited into the start control node of the ICN. The token game of an ICN is similar to the token game of a Petri Net. An activity node is *enabled* if it has at least on token in its input node; this means that the activity may be enacted. An enabled activity node *starts firing* when the activity is enacted. In this case, a single token is removed from the input node, and is placed inside of the activity node. The work item created as the result of the activity enactment is attached to the token. When the activity ends (normally or not),) the work item is dis-associated from the activity token, and the activity node *ends firing*: the token is removed from the activity node, and placed into the output node of the activity node.

And-nodes may be viewed in the context of this work as instantaneous Petri nets transitions with multiple input/output places. An and-node is enabled if each of its input nodes has at least one token. In this case, the node fires by *consuming* one single token from each of its input nodes, and producing one single token into each of its output nodes.

An or-node is enabled if at least one of its input places has a token. In this case, one token is selected from one input node, and it is deposited into the or-node. Then, the test associated with the or-node, if any, is executed and the outcome is used to select which output node the token moves to. If several outgoing edges match the outcome of the test, then the choice may be based on priority attached to the outgoing edges. The test associated with an or-node may be based on application or process data. For example, the test associated with p_4 in DI_Process:1.0 is based on external events (the customer arrives or calls to cancel her reservation.)

The execution of a process ends when its marking (i.e. its token distribution) consists of a single token residing in the exit place. Several executions of the same activity may be simultaneously in progress: this is reflected by the existence of several tokens residing within its (activity) node.

7.2 The change

And the troubles begin... On one hand, a wave of reservation requests, reservation losses and cancellations hits *The Desert Inn*. The staff is overwhelmed by the volume of telephone calls they have to deal with, causing considerable confirmation delays and missed business opportunities. Dissatisfied customers and travel agencies cancel their reservations and refuse to pay the cancellation fee. On the other hand, a serious bug in the billing system raises some concerns directed toward the IT department of the hotel. To respond, the management staff decides to take the following measures:

- Deploy a series of upgraded systems including *EZ_Bill* for billing, *EZ_Pay* for payment processing, *EZ_Res* for reservation online processing, *EZ_In* for speedier check in, *EZ_Out* for smooth check out, and *EZ_Cancel* to deal with the cancellation process. *EZ_Res* is a secured online fully automated reservation system with enhanced voice coaching capabilities.

- All *VIC* reservation requests are to be handled by *EZ_Res*.

- Every *VIC* must pay a deposit, the equivalent of one night stay, in advance (i.e. prior to check in.) This transaction is conducted on-line by the *EZ_Res* system. A confirmation message (with a reservation number and a credit card approval number,) or a rejection message, is either voice-read (if the customer remains on line,) or fax-ed (if the customer chooses so.)

- Unless an advance delay notification is received by the reception staff, the latest guaranteed check in time is 3.00 p.m.

- A highly trained team of agents, *The Desert Inn Club* team, is dedicated to handle the *HIC*s.

- Every customer pays his/her hotel bill before leaving.

- A new discounted plan, *The Desert Inn Passport (DIP)*, with some restrictions including 6-week advanced booking-and-payment and 72-hour advanced-cancellation restrictions, is to be introduced within 6 weeks.

- A pro-rated cancellation refund policy is applied to all canceled reservations.

- The collection agency is excused.

All new cases are to be handled in accordance with these measures. The new version of this process, referred to as *DI_Process:2.0*, is depicted in figure 3. The automated activity *EZ_Greet* greets the customer, prompts and collects some preliminary information. The *DI_Club* activity models the processing of a *HIC* case by a member of *The Desert Inn Club* team. The *No_Show* activity is activated when a customer fails to check in before 3.00 p.m.

7.3 The change policy

These measures are critical, therefore effective immediately. The change policy, referred to as the *Reservation Change Policy*, is stated as follows:

- All future reservations have to be processed according to the new procedure.
- The old billing and payment systems are fixed and maintained for one month.
- The billing and the payment of the current cases can be handled either by the new systems, or the the patched old systems.
- The confirmed *HIC*s who have not yet checked in must be handled by the *Desert Inn* team who will decide the appropriate measures to take.
- The pay-before-you-leave policy must not apply to to the currently checked in *HIC*s.
- All the confirmed *VIC*s who are expected to check-in within two weeks, must transition to the new procedure before they check-in. The others must be sent a notification letter, to bring their attention to the hotel's change of policy, the availability of the discount program, and the delay within which they must respond. The delay period takes effect in two weeks and ends four weeks thereafter. It must be selected so as to not overwhelm the system and to give a customer the opportunity to take advantage of the new discount program. To deal with the expected volume of responses, a light version of *EZ_Res*, referred to as *Express Res* is deployed immediately.
- The pending reservations, once they are accepted, should be treated the same way as the confirmed reservations, as previously outlined. Moreover, the confirmation letters may be sent by the new procedure only.

7.4 The change process

The change process represented by the ICN depicted in figure 4 deals with the current reservation cases which are not canceled. The change meta-activities are represented in gray-colored ellipses. The change process proceeds as follows:

First, a test is performed on the case using the data in the reservation form to determine if it is associated with a confirmed *VIC*, a *VIC* whose reservation is still being processed, or a *HIC*.

In the first case, If the customer has not checked-in and if she is expected to check-in within 2 weeks, then two bridges (a bridge may be viewed as an activity which consumes tokens from the old procedure, and produces tokens in the new procedure) are established: the first bridge links p_4 to q_4, and the second bridge links p_2 to q_2. The first bridge takes care of the situation where the reservation is confirmed, and the customer has already been notified; it allows the only case token which resides in p_4 to move to q_4 in the new procedure. The second bridge takes care of the situation where the reservation is confirmed, but the customer has not been notified; the only case token which resides in p_2 moves to q_2.

If the customer is expected to check-in after two weeks, then the case tokens are flushed; this action may be done manually or automatically by the system, then the customer is notified by the change activity *Notify*, then a single reminder is set. If the delay expires, then a token is *injected* into the new procedure at q_2. Here, is is assumed that the *Notify* activity changes the status of the reservation to 'Rejected'. If the customer responds within the delay, then the activity *Express Res* is activated, and then the case is moved to the new procedure at q_2. In both situations, the case resumes in the new procedure at q_2.

If the customer has already checked-in, then a series of bridge is established to catch the case before it gets to check-out. This bridges ensure that any *VIC* who has not yet checked in is properly billed by the new procedure.

In the second case, the change process waits for the completion of the batch processing. If the reservation is confirmed, then the case is treated as outlined earlier. Else, the case remains in the old procedure.

In the third case (i.e. *HIC* case,) a test is performed on the case to check if the customer has checked-in. If yes, then the only case token in the old procedure is moved from p_4 to q_1, so that the reservation is handled by the *DI_Club* team. If no, the change process waits until the customer checks out (to avoid applying the pay-before-you-go policy,) then the case tokens are flushed, then the customer is billed according to the old fixed system, then the case is moved to the new procedure after the customer pays the bill. Note here, that during the time period which starts after the tokens are flushed, and ends when the customer pays the bill, the case is running neither in the old procedure, nor in the new procedure, but in the change process. Although, this case study does not illustrate it, a case may be running simultaneously in three procedures; the old procedure, the new procedure and the change procedure.

8 Conclusions

In this paper, we have presented $\mathcal{ML}$-$\mathcal{DEWS}$, a modeling language for the specification of change. The underlying philosophy of this language is based upon the observation that a change is a process itself. This language allows a user to describe change policies which address the many crucial issues related to procedure changes. Its unified approach deals conveniently with schema and instance changes (e.g. exceptions.) We have also illustrated through examples and the case study the usefulness and the uniqueness of this language.

$\mathcal{ML}$-$\mathcal{DEWS}$ is not 'yet another language'; it can be used either as is, or as a basis for other change models. Currently, it is being used to specify procedure changes within the University of Colorado, and future plans to use it within a telephone company call center are underway.

References

1. A. Agostini, F. De Michelis. "Simple workflow models" In Proceedings of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
2. Apel, U., "On-line Software Extension and Modification" in Electrical Communications, V.64, N.4 (1991).
3. E. Badouel, J. Oliver. "Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes" In Proceedings of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
4. P. Bichler, G.. Preuner, M. Schrefl. "Workflow Transparency". In Proc on the Intl'l Conf. on Advanced Information Systems (CAiSE 97).
5. U. M. Borghoff, P. Bottoni, P. Mussio, R.Pareschi. "Reflective Agents for Adaptive Workflows" In Proc.. Ent Int'l Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'97), April 21-23 1997, London, U. K.
6. F. Casati, S. Ceri, B. Pernici, G. Pozzi. "Workflow Evolution". In Proc. of the 15th International Conference on Conceptual Modeling (OOER 96), Cottbus, Germany.
7. G. De Michelis, C. A. Ellis. Computer Supported Cooperative Work and Petri Nets. Third Advanced Course on Petri Nets, Dagstuhl Castle, Germany (1996). Springer Verlag Lecture Notes in Computer Science.

8. C. A. Ellis and G. J. Nutt. "Modeling and Enactment of Workflow Systems". In M. Ajmone Marsan, editor, Application and Theory of Petri Nets 1993, volume 691 of Lecture Notes in Computer Science, pages 1-16. Springer-Verlag, Berlin, 1993.
9. C.A. Ellis, K. Keddara, G. Rozenberg. "Dynamic Change within Workflow Systems". In Proc. of the Conference on organizational Computing systems, ACM Press, New York (1995) 10-21.
10. C. A. Ellis, K. Keddara, J. Wainer. "Modeling Dynamic Change Using Timed Hybrid Flow Nets" In Proc. of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
11. C. A. Ellis and G. J. Nutt. "Office Information Systems and Computer Science". In ACM Computer Survey, vol. 12, no 1, March 1980.
12. IBM, "The IBM Flowmark System: Modeling Workflow, Version 2, Release 2" Publication No.SH-19-8241-01, 1996.
13. Jablonski S., Bussler C. "Workflow Management, Modeling Concepts, Architecture and Implementation". 1996.
14. K. Keddara. "On the Dynamic Evolution of Workflow Systems". Ph.D Thesis, University of Colorado, Boulder, USA. In progress.
15. Klein, M., (Ed.) "Toward Adaptive Workflow Systems" Workshop at the ACM CSCW'98 Conference, Seattle, WA, August, 1998.
16. Kriefelts T et al. "DOMINO: A System for the Specification and the Automation of Cooperative Office Processes". In proceedings of the EuroMicro84, edited by Wilson and Myrhaug.
17. Kumar, A., Zhao, L., "A Framework for Dynamic Routing and Operational Integrity in a Workflow Management System" University of Colorado Report, 1997.
18. Lee, J., et.al., "The PIF Process Interchange Format and Framework, Version 1.1" Proceedings of the Workshop on Ontological Engineering, ECAI'96, Budapest, Hungary, 1996.
19. Meirs, D., et.al., "Work Management Technologies Report" Process Products Watch, Enix Limited, 1998.
20. Malone, T., et.al., "Tools for Inventing Organizations: Handbook of Organizational Processes" Management Science Journal, August 1998.
21. Osterweil, L., "Automated Support for the Enactment of Rigorously Described Software Processes", In proceedings of the Third International Process Programming Workshop, 1988, pp. 122-125. IEEE Computer Society Press.
22. Reichert, M., Dadam, P., "Supporting Dynamic Changes of Workflows Without Loosing Control" Journal of Intelligent Information Systems, V.10, N.2, 1998.
23. H. Saastamoinen "On the Handling of Exceptions in Information Systems" University of Jyvaskyla PhD Dissertation, Nov. 1995.
24. Star, S. L., and Ruhleder, K., "The Ecology of Infrastructures: Problems in the Implementation of Large Scale Information Systems," Information Systems Research 7(1), pp.111-134, 1996.
25. Stotts, P. D., et.al., "Modelling the Logical Structure of Flexible Manufacturing Systems with Petri Nets" in Computer Communications, V.12, N.8 (U.K.) August, 1989.
26. Strong, D., and Miller, S., "Exceptions and Exception Handling in Computerized Information Processes" ACM TOIS, V.13, N.2. April, 1995.
27. Suchman, L., "Plans and Situated Actions: The Problem of Human - Machine Communication. Cambridge University Press. Cambridge, England. 1987.
28. W.M.P. van der Aalst. "Verification of Workflow Nets". In P. Azema and G. Balbo, editors, Application and Theory of Petri Nets 1997,volume 1248 of Lecture Notes in Computer Science, pages 407-426. Springer-Verlag, Berlin, 1997.
29. W.M.P. van der Aalst. "Finding Errors in the Design of a Workflow Process". In Proc. of WFM98: Workflow Management: Net-Based Concepts, Models, Techniques and Tools, PN98, Lisbon, Portugal.
30. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011) Technical Report, Workflow Management Coalition, Brussels, 1996.

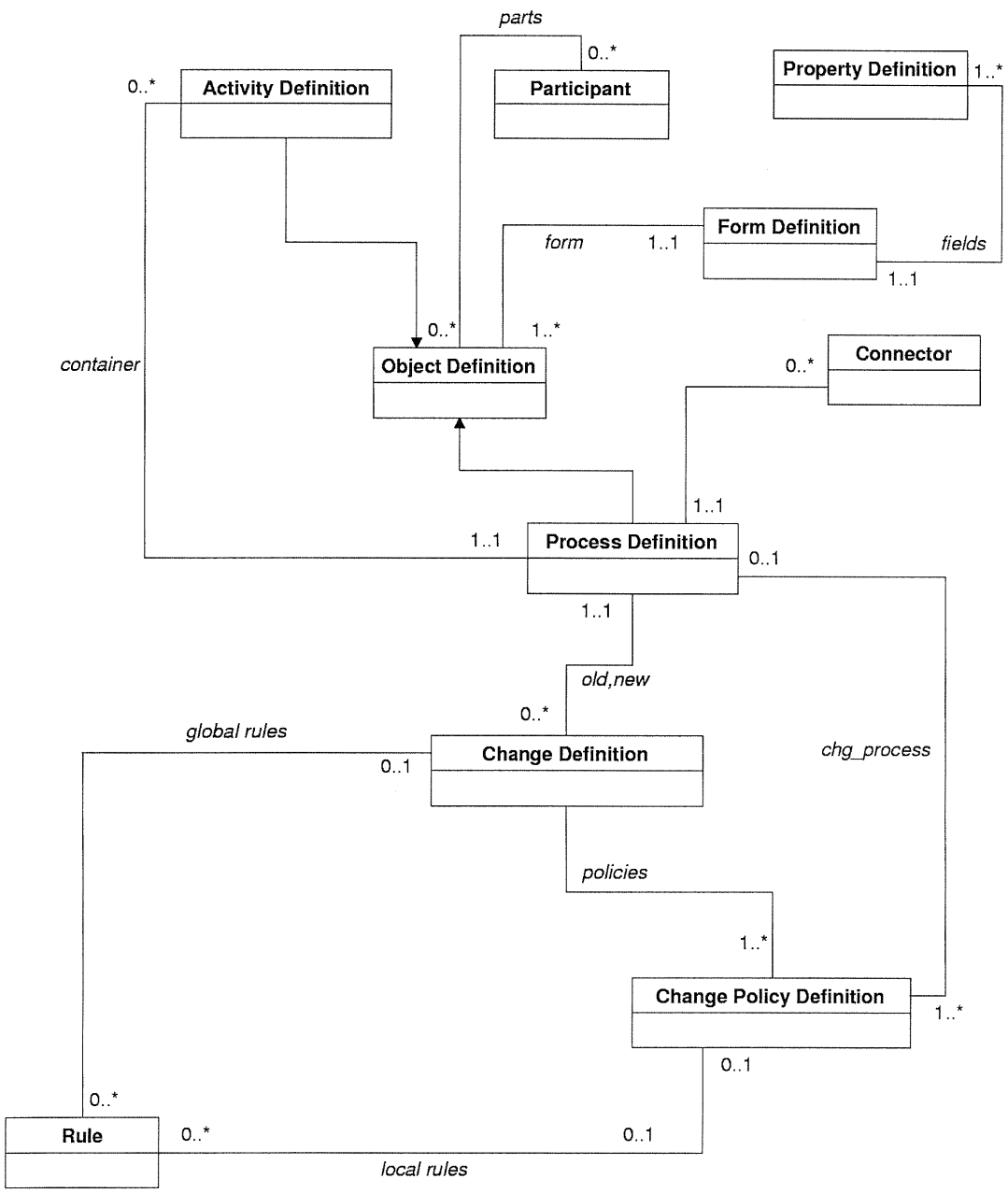


Fig. 1. The definition and the change specification model

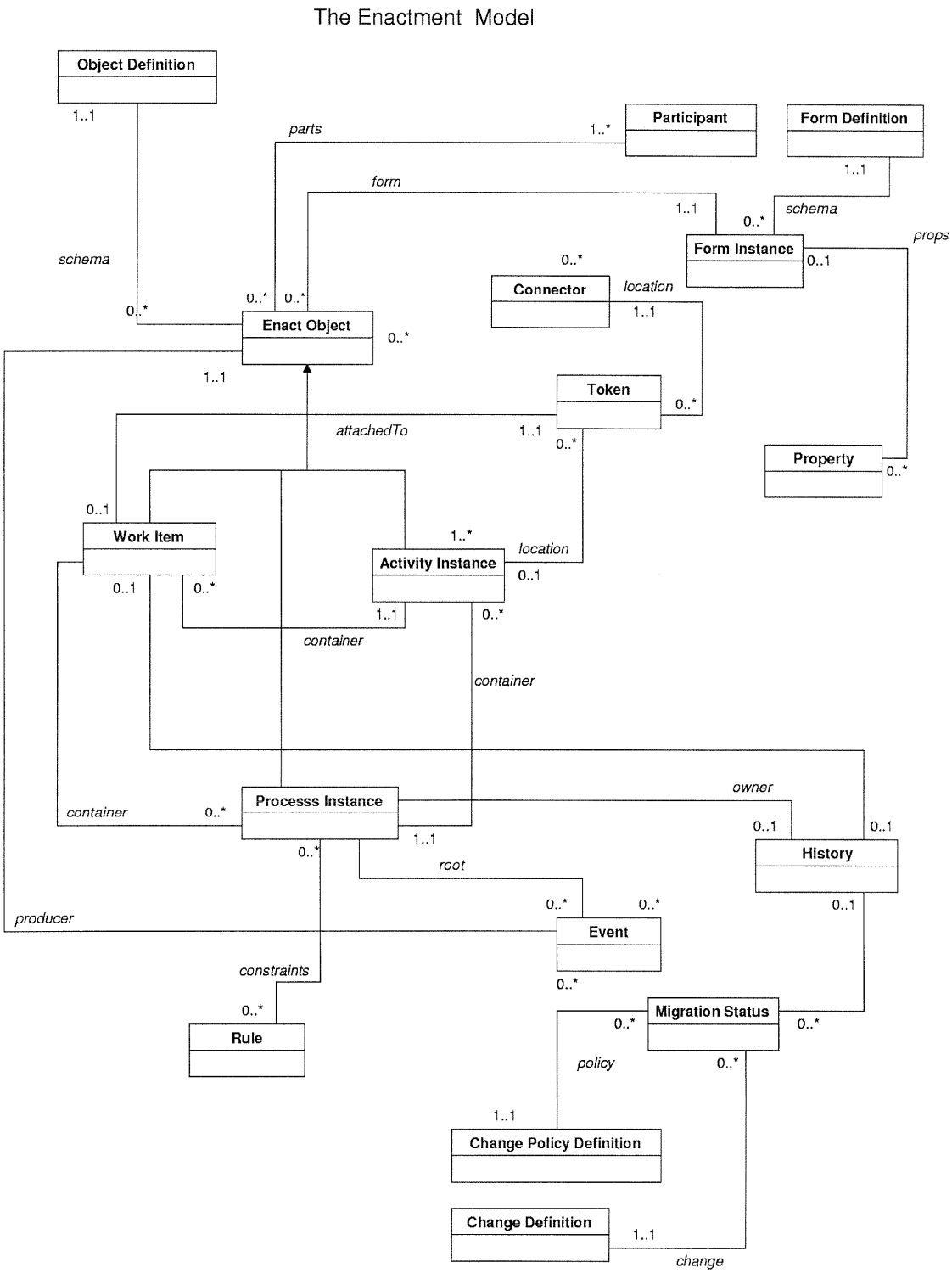


Fig. 2. The enactment model

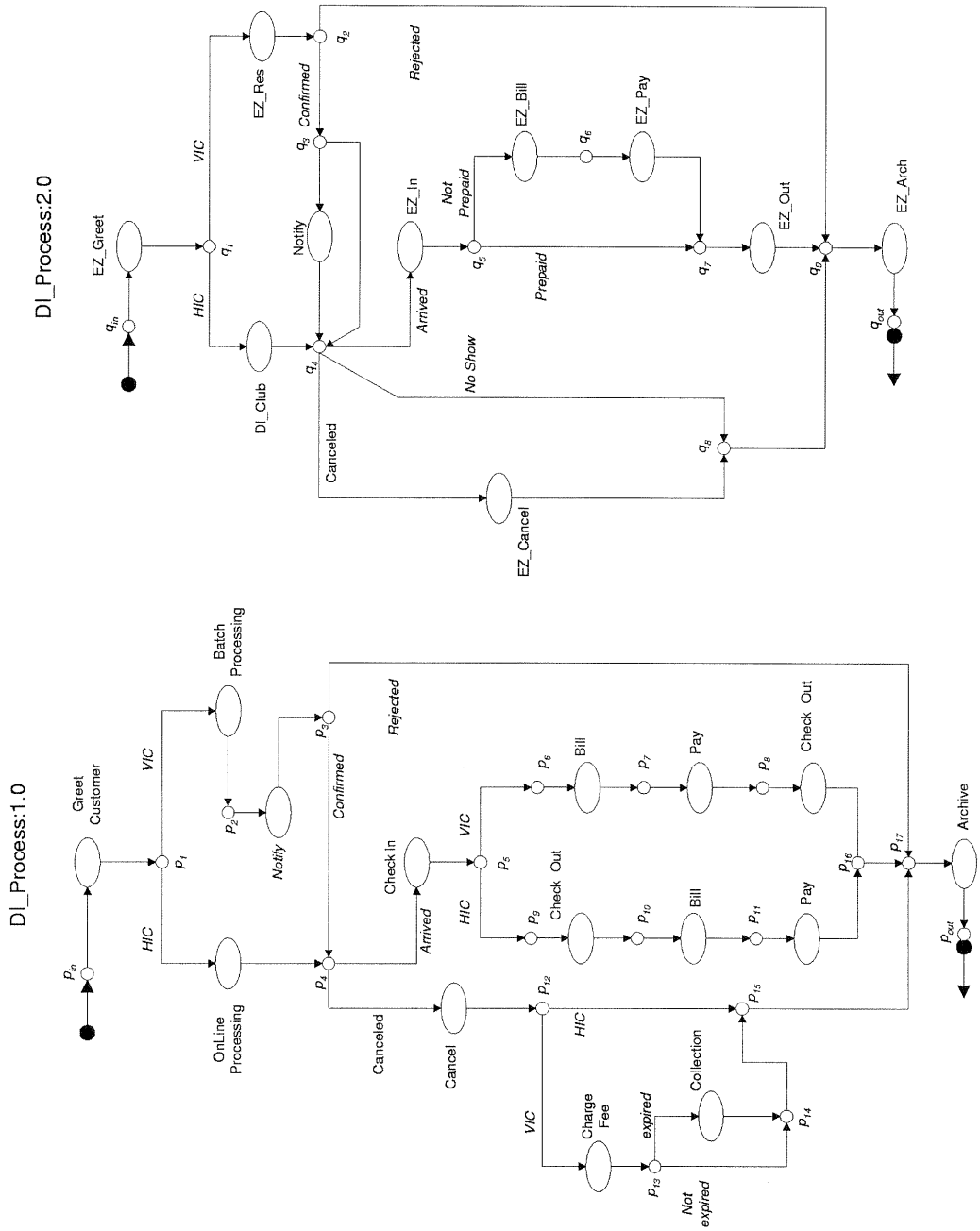


Fig. 3. Two versions of the *Desert Inn* reservation procedures